```c
#include <stdio.h>

void mergeSort(int arr[], int l, int r);
void merge(int arr[], int l, int m, int r);

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;

    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;

        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

int main()
```

```c
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array: \n");
    for (int i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

```
Sorted array:
5 6 7 11 12 13

Process returned 0 (0x0)   execution time : 0.028 s
Press any key to continue.
```

```c
#include <stdio.h>

void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: \n");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

```
Sorted array:
1 5 7 8 9 10

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <stdlib.h>

struct Term
{
    int coeff;
    int exp;
    struct Term* next;
};

struct Term* createTerm(int coeff, int exp)
{
    struct Term* newTerm = (struct Term*)malloc(sizeof(struct Term));
    newTerm->coeff = coeff;
    newTerm->exp = exp;
    newTerm->next = NULL;
    return newTerm;
}

void insertTerm(struct Term** poly, int coeff, int exp)
{
    struct Term* newTerm = createTerm(coeff, exp);
    if (*poly == NULL || (*poly)->exp < exp)
    {
        newTerm->next = *poly;
        *poly = newTerm;
    }
    else
    {
        struct Term* temp = *poly;
        while (temp->next && temp->next->exp > exp)
        {
            temp = temp->next;
        }
        if (temp->exp == exp)
        {
            temp->coeff += coeff;
            free(newTerm);
        } else
        {
            newTerm->next = temp->next;
            temp->next = newTerm;
        }
    }
}

struct Term* addPolynomials(struct Term* poly1, struct Term* poly2)
{
    struct Term* result = NULL;
    while (poly1 && poly2)
    {
        if (poly1->exp > poly2->exp)
        {
            insertTerm(&result, poly1->coeff, poly1->exp);
            poly1 = poly1->next;
        }
        else if (poly1->exp < poly2->exp)
        {
            insertTerm(&result, poly2->coeff, poly2->exp);
            poly2 = poly2->next;
        }
        else
        {
            insertTerm(&result, poly1->coeff + poly2->coeff, poly1->exp);
            poly1 = poly1->next;
            poly2 = poly2->next;
```

```c
        }
    }
    while (poly1)
    {
        insertTerm(&result, poly1->coeff, poly1->exp);
        poly1 = poly1->next;
    }
    while (poly2)
    {
        insertTerm(&result, poly2->coeff, poly2->exp);
        poly2 = poly2->next;
    }
    return result;
}

void displayPolynomial(struct Term* poly)
{
    while (poly)
    {
        printf("%dx^%d ", poly->coeff, poly->exp);
        poly = poly->next;
        if (poly) printf("+ ");
    }
    printf("\n");
}

int main()
{
    struct Term* poly1 = NULL, *poly2 = NULL, *result = NULL;
    int n1, n2, coeff, exp;

    // Input for the first polynomial
    printf("Enter the number of terms in the first Polynomial: ");
    scanf("%d", &n1);
    for (int i = 0; i < n1; ++i)
   {
        printf("Enter coefficient and exponent for term %d: ", i + 1);
        scanf("%d %d", &coeff, &exp);
        insertTerm(&poly1, coeff, exp);
    }

    // Input for the second polynomial
    printf("\nEnter the number of terms in the second Polynomial: ");
    scanf("%d", &n2);
    for (int i = 0; i < n2; ++i)
   {
        printf("Enter coefficient and exponent for term %d: ", i + 1);
        scanf("%d %d", &coeff, &exp);
        insertTerm(&poly2, coeff, exp);
    }

    result = addPolynomials(poly1, poly2);
    printf("Resultant Polynomial: ");
    displayPolynomial(result);
    return 0;
}
```

```
Enter the number of terms in the first Polynomial: 3
Enter coefficient and exponent for term 1: 5 3
Enter coefficient and exponent for term 2: 3 2
Enter coefficient and exponent for term 3: 3 0

Enter the number of terms in the second Polynomial: 3
Enter coefficient and exponent for term 1: 4 3
Enter coefficient and exponent for term 2: 6 1
Enter coefficient and exponent for term 3: 1 0
Resultant Polynomial: 9x^3 + 3x^2 + 6x^1 + 4x^0

Process returned 0 (0x0)   execution time : 87.015 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <ctype.h>

int prec(char c)
{
    switch (c)
    {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
        default:
            return -1;
    }
}

void infixToPostfix(char* infix, char* postfix)
{
    int i, j;
    char stack[20];
    int top = -1;

    for (i = 0, j = 0; infix[i] != '\0'; i++)
    {
        if (isalnum(infix[i]))
        {
            postfix[j++] = infix[i];
        }
        else if (infix[i] == '(')
        {
            stack[++top] = infix[i];
        }
        else if (infix[i] == ')')
        {
            while (top > -1 && stack[top] != '(')
            {
                postfix[j++] = stack[top--];
            }
            if (top > -1 && stack[top] != '(')
            {
                return;
            }
            top--;
        }
        else
        {
            while (top > -1 && prec(stack[top]) >= prec(infix[i]))
            {
                postfix[j++] = stack[top--];
            }
            stack[++top] = infix[i];
        }
    }

    while (top > -1)
    {
        postfix[j++] = stack[top--];
    }
    postfix[j] = '\0';
}

int main()
```

```
{
    char infix[50], postfix[50];
    printf("Enter infix expression: ");
    scanf("%s", infix);
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    return 0;
}
```

```
Enter infix expression: (8+2)*(7-3)/4
Postfix expression: 82+73-*4/

Process returned 0 (0x0)   execution time : 18.563 s
Press any key to continue.
```

```c
#include<stdio.h>
#include<stdlib.h>

struct Node {
    int data;
    struct Node *left, *right;
};

struct Node* create(int value) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = value;
    node->left = node->right = NULL;
    return node;
}

struct Node* insert(struct Node* root, int value) {
    if (!root) return create(value);
    if (value < root->data) root->left = insert(root->left, value);
    else if (value > root->data) root->right = insert(root->right, value);
    return root;
}

void inorder(struct Node* root) {
    if (root) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

int main() {
    struct Node* root = NULL;
    int n, value;
    printf("Enter the number of elements to insert: ");
    scanf("%d", &n);
    root = insert(root, 30);  // Insert the root first
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &value);
        insert(root, value);
    }
    printf("In-order traversal: ");
    inorder(root);
    printf("\n");
    return 0;
}
```

C:\Users\hp\Downloads\dsa\prog4.exe

```
Enter the number of elements to insert: 4
Enter 4 elements:
25
40
10
15
In-order traversal: 10 15 25 30 40

Process returned 0 (0x0)   execution time : 14.244 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <stdlib.h>

// Structure for a tree node
struct Node
{
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// Function prototypes
int max(int a, int b);
int height(struct Node *node);
int getBalance(struct Node *node);
struct Node *newNode(int key);
struct Node *rightRotate(struct Node *y);
struct Node *leftRotate(struct Node *x);
struct Node *insert(struct Node *node, int key);
void inOrderTraversal(struct Node *root);

int main()
{
    struct Node *root = NULL;
    int n, value;

    // Step 6: Take user input for number of elements and their values
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    printf("Enter the elements:\n");

    // Step 7: Iterate to insert each value into the AVL tree
    for (int i = 0; i < n; ++i)
    {
        scanf("%d", &value);
        root = insert(root, value); // Insert each value into the AVL Tree
    }

    // Step 8: Display the in-order traversal of the AVL tree
    printf("In-order Traversal of AVL Tree: ");
    inOrderTraversal(root);
    printf("\n");

    return 0;
}

// Step 3: Utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b) ? a : b;
}

// Step 3: Utility function to get the height of a node
int height(struct Node *node)
{
    if (node == NULL)
        return 0;
    return node->height;
}

// Step 3: Utility function to get the balance factor of a node
int getBalance(struct Node *node)
{
    if (node == NULL)
        return 0;
```

```c
    return height(node->left) - height(node->right);
}

// Step 3: Function to create a new node with the given key
struct Node *newNode(int key)
{
    struct Node *node = (struct Node *)malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // New node is initially at height 1
    return node;
}

// Step 4: Function to perform right rotation on the given node
struct Node *rightRotate(struct Node *y)
{
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}

// Step 4: Function to perform left rotation on the given node
struct Node *leftRotate(struct Node *x)
{
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Return new root
    return y;
}

// Step 5: Function to insert a key into the AVL tree
struct Node *insert(struct Node *node, int key)
{
    // Step 5: Perform standard BST insert
    if (node == NULL)
        return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Duplicate keys are not allowed
        return node;

    // Step 5: Update height of this ancestor node
    node->height = 1 + max(height(node->left), height(node->right));
```

```c
    // Step 5: Get the balance factor to check if this node became unbalanced
    int balance = getBalance(node);

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    // Return the unchanged node pointer
    return node;
}

// Step 8: Function to perform in-order traversal of the AVL tree
void inOrderTraversal(struct Node *root)
{
    if (root != NULL)
    {
        inOrderTraversal(root->left);
        printf("%d ", root->key);
        inOrderTraversal(root->right);
    }
}
```

C:\Users\hp\Downloads\Files1\prog5.exe

```
Enter the number of elements: 5
Enter the elements:
7 8 9 3 1
In-order Traversal of AVL Tree: 1 3 7 8 9

Process returned 0 (0x0)   execution time : 9.924 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_HEAP_SIZE 100

struct BinaryHeap {
    int arr[MAX_HEAP_SIZE];
    int size;
};

struct BinaryHeap* createHeap() {
    struct BinaryHeap* heap = (struct BinaryHeap*)malloc(sizeof(struct BinaryHeap));
    heap->size = 0;
    return heap;
}

void insert(struct BinaryHeap* heap, int value) {
    if (heap->size == MAX_HEAP_SIZE) return;
    int i = heap->size++, parent;
    while (i > 0 && value < heap->arr[parent = (i - 1) / 2]) {
        heap->arr[i] = heap->arr[parent];
        i = parent;
    }
    heap->arr[i] = value;
}

int extractMin(struct BinaryHeap* heap) {
    if (heap->size == 0) return -1;
    int min = heap->arr[0], i = 0;
    heap->arr[0] = heap->arr[--heap->size];
    while (1) {
        int left = 2*i + 1, right = 2*i + 2, smallest = i;
        if (left < heap->size && heap->arr[left] < heap->arr[smallest]) smallest = left;
        if (right < heap->size && heap->arr[right] < heap->arr[smallest]) smallest = right;
        if (smallest != i) {
            int temp = heap->arr[i];
            heap->arr[i] = heap->arr[smallest];
            heap->arr[smallest] = temp;
            i = smallest;
        } else break;
    }
    return min;
}

int main() {
    struct BinaryHeap* heap = createHeap();
    int n, value;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &value);
        insert(heap, value);
    }
    printf("Sorted elements: ");
    for (int i = 0; i < n; i++) printf("%d ", extractMin(heap));
    printf("\n");
    return 0;
}
```

C:\Users\hp\Downloads\dsa\prog6.exe

```
Enter the number of elements: 5
8 3 5 1 4
Sorted elements: 1 3 4 5 8

Process returned 0 (0x0)   execution time : 9.730 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>

struct Node
{
    int key;
    int degree;
    struct Node* parent;
    struct Node* child;
    struct Node* left;
    struct Node* right;
    int marked;
};

struct FibonacciHeap
{
    struct Node* min;
    int numNodes;
};

// Function prototypes
struct FibonacciHeap* createFibonacciHeap();
struct Node* createNode(int key);
void insertNode(struct FibonacciHeap* heap, struct Node* node);
struct Node* mergeHeaps(struct Node* a, struct Node* b);
struct Node* consolidate(struct Node* minNode, int maxDegree);
struct Node* extractMin(struct FibonacciHeap* heap);
void freeNode(struct Node* node);

int main()
{
    struct FibonacciHeap* fibHeap = createFibonacciHeap();
    int n, key;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    printf("Enter the elements:\n");
    for (int i = 0; i < n; ++i)
    {
        scanf("%d", &key);
        struct Node* node = createNode(key);
        insertNode(fibHeap, node);
    }

    // Extract minimum element
    struct Node* minNode = extractMin(fibHeap);
    if (minNode)
    {
        printf("Minimum element extracted: %d\n", minNode->key);
        freeNode(minNode);
    }
    else
    {
        printf("Heap is empty.\n");
    }

    return 0;
}

struct FibonacciHeap* createFibonacciHeap()
{
    struct FibonacciHeap* heap = (struct FibonacciHeap*)malloc(sizeof(struct FibonacciHeap));
    heap->min = NULL;
```

```c
    heap->numNodes = 0;
    return heap;
}

struct Node* createNode(int key)
{
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->key = key;
    node->degree = 0;
    node->parent = NULL;
    node->child = NULL;
    node->left = node;
    node->right = node;
    node->marked = 0;
    return node;
}

void insertNode(struct FibonacciHeap* heap, struct Node* node)
{
    if (heap->min == NULL)
    {
        heap->min = node;
    }
    else
    {
        heap->min->left->right = node;
        node->left = heap->min->left;
        heap->min->left = node;
        node->right = heap->min;
        if (node->key < heap->min->key)
        {
            heap->min = node;
        }
    }
    heap->numNodes++;
}

struct Node* mergeHeaps(struct Node* a, struct Node* b)
{
    if (a == NULL) return b;
    if (b == NULL) return a;

    struct Node* temp = a->right;
    a->right = b->right;
    b->right->left = a;
    b->right = temp;
    temp->left = b;

    return (a->key < b->key) ? a : b;
}

struct Node* extractMin(struct FibonacciHeap* heap)
{
    struct Node* minNode = heap->min;
    if (minNode == NULL) return NULL;

    struct Node* child = minNode->child;
    if (child != NULL)
    {
        struct Node* temp = child;
        do
        {
            temp->parent = NULL;
            temp = temp->right;
        } while (temp != child);
```

```c
        struct Node* leftNode = minNode->left;
        struct Node* rightNode = minNode->right;

        leftNode->right = child;
        child->left->right = rightNode;
        rightNode->left = child->left;
        child->left = leftNode;
    }

    minNode->left->right = minNode->right;
    minNode->right->left = minNode->left;

    if (minNode == minNode->right)
    {
        heap->min = NULL;
    }
    else
    {
        heap->min = minNode->right;
        heap->min = consolidate(heap->min, (int)(log2(heap->numNodes)) + 1);
    }

    heap->numNodes--;
    return minNode;
}

struct Node* consolidate(struct Node* minNode, int maxDegree)
{
    struct Node** degreeTable = (struct Node**)calloc(maxDegree + 1, sizeof(struct Node*));
    struct Node* current = minNode;

    do
    {
        int degree = current->degree;
        while (degreeTable[degree] != NULL)
        {
            struct Node* other = degreeTable[degree];
            if (current->key > other->key)
            {
                struct Node* temp = current;
                current = other;
                other = temp;
            }
            other->left->right = other->right;
            other->right->left = other->left;

            other->parent = current;
            if (current->child == NULL)
            {
                current->child = other;
                other->right = other;
                other->left = other;
            }
            else
            {
                other->left = current->child;
                other->right = current->child->right;
                current->child->right->left = other;
                current->child->right = other;
            }

            current->degree++;
            degreeTable[degree] = NULL;
            degree++;
        }
        degreeTable[degree] = current;
```

```
        current = current->right;
    } while (current != minNode);

    struct Node* newMin = NULL;
    for (int i = 0; i <= maxDegree; ++i)
    {
        if (degreeTable[i] != NULL)
        {
            if (newMin == NULL || degreeTable[i]->key < newMin->key)
            {
                newMin = degreeTable[i];
            }
        }
    }

    free(degreeTable);
    return newMin;
}

void freeNode(struct Node* node)
{
    if (!node) return;

    if (node->child)
    {
        struct Node* child = node->child;
        do
        {
            struct Node* nextChild = child->right;
            freeNode(child);
            child = nextChild;
        } while (child != node->child);
    }

    free(node);
}
```

C:\Users\hp\Downloads\Files1\prog7.exe

```
Enter the number of elements: 5
Enter the elements:
4 8 2 10 6
Minimum element extracted: 2

Process returned 0 (0x0)   execution time : 13.517 s
Press any key to continue.
```

```c
#include <stdio.h>
 #include <stdlib.h>
 #define MAX_VERTICES 100
 // Adjacency list node
 struct Node
 {
    int data;
    struct Node* next;
 };
 // Graph representation using adjacency list
 struct Graph
 {
    int numVertices;
    struct Node* adjList[MAX_VERTICES];
 };
 // Function prototypes
 struct Graph* createGraph(int vertices);
 void addEdge(struct Graph* graph, int src, int dest);
 void printGraph(struct Graph* graph);
 void DFS(struct Graph* graph, int startVertex);
 void BFS(struct Graph* graph, int startVertex);
 // Main function
 int main()
 {
    int vertices, edges;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    struct Graph* graph = createGraph(vertices);
    printf("Enter the number of edges: ");
    scanf("%d", &edges);
    printf("Enter edges (source destination):\n");
    for (int i = 0; i < edges; ++i)
    {
        int src, dest;
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }
    printf("Graph representation:\n");
    printGraph(graph);
    int startVertex;
    printf("Enter the starting vertex for traversals: ");
    scanf("%d", &startVertex);
    printf("\nDepth-First Search (DFS) starting from vertex %d:\n", startVertex);
    DFS(graph, startVertex);
    printf("\nBreadth-First Search (BFS) starting from vertex %d:\n", startVertex);
    BFS(graph, startVertex);
    return 0;
}
 // Function to create a graph with a given number of vertices
 struct Graph* createGraph(int vertices)
 {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    for (int i = 0; i < vertices; ++i)
    {
        graph->adjList[i] = NULL;
    }
    return graph;
 }
 // Function to add an edge to the graph
 void addEdge(struct Graph* graph, int src, int dest)
 {
    // Add edge from src to dest
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = dest;
    newNode->next = graph->adjList[src];
```

```c
    graph->adjList[src] = newNode;
    // Add edge from dest to src (for undirected graph)
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = src;
    newNode->next = graph->adjList[dest];
    graph->adjList[dest] = newNode;
}
// Function to print the graph
void printGraph(struct Graph* graph)
{
    for (int i = 0; i < graph->numVertices; ++i)
    {
        struct Node* current = graph->adjList[i];
        printf("Adjacency list for vertex %d: ", i);
        while (current)
        {
            printf("%d -> ", current->data);
            current = current->next;
        }
        printf("NULL\n");
    }
}
// Recursive function for Depth-First Search (DFS)
void DFSUtil(struct Graph* graph, int vertex, int visited[])
{
    visited[vertex] = 1;
    printf("%d ", vertex);
    struct Node* current = graph->adjList[vertex];
    while (current)
    {
        if (!visited[current->data])
        {
            DFSUtil(graph, current->data, visited);
        }
        current = current->next;
    }
}
// Depth-First Search (DFS) traversal
void DFS(struct Graph* graph, int startVertex)
{
    int visited[MAX_VERTICES] = {0};
    DFSUtil(graph, startVertex, visited);
}
// Breadth-First Search (BFS) traversal
void BFS(struct Graph* graph, int startVertex)
{
    int visited[MAX_VERTICES] = {0};
    int queue[MAX_VERTICES];
    int front = 0, rear = 0;
    visited[startVertex] = 1;
    queue[rear++] = startVertex;
    while (front < rear)
    {
        int currentVertex = queue[front++];
        printf("%d ", currentVertex);
        struct Node* current = graph->adjList[currentVertex];
        while (current)
        {
            if (!visited[current->data])
            {
                visited[current->data] = 1;
                queue[rear++] = current->data;
            }
            current = current->next;
        }
    }
```

```
}
```

```
Enter the number of vertices: 5
Enter the number of edges: 6
Enter edges (source destination):
0 1
0 2
1 3
1 4
2 4
3 4
Graph representation:
Adjacency list for vertex 0: 2 -> 1 -> NULL
Adjacency list for vertex 1: 4 -> 3 -> 0 -> NULL
Adjacency list for vertex 2: 4 -> 0 -> NULL
Adjacency list for vertex 3: 4 -> 1 -> NULL
Adjacency list for vertex 4: 3 -> 2 -> 1 -> NULL
Enter the starting vertex for traversals: 0

Depth-First Search (DFS) starting from vertex 0:
0 2 4 3 1
Breadth-First Search (BFS) starting from vertex 0:
0 2 1 4 3
Process returned 0 (0x0)   execution time : 32.530 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <stdlib.h>

// Structures for edges, graph, and subsets
struct Edge
{
    int source, destination, weight;
};
struct Graph
{
    int V, E;
    struct Edge* edges;
};
struct Subset
{
    int parent,rank;
};

// Function prototypes
struct Graph* createGraph(int V, int E);
int find(struct Subset subsets[], int i);
void unionSets(struct Subset subsets[], int x, int y);
int compareEdges(const void* a, const void* b);
void kruskal(struct Graph* graph);

int main()
{
    int V, E;
    // Input number of vertices and edges
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    printf("Enter the number of edges: ");
    scanf("%d", &E);
    struct Graph* graph = createGraph(V, E);

    // Input edges
    printf("Enter edges (source destination weight):\n");
    for (int i = 0; i < E; ++i)
    {
        scanf("%d %d %d", &graph->edges[i].source, &graph->edges[i].destination, &graph->edges[i].weight);
    }

    // Find and print MST
    kruskal(graph);

    // Free allocated memory
    free(graph->edges);
    free(graph);
    return 0;
}

// Create a graph
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edges = (struct Edge*)malloc(E * sizeof(struct Edge));
    return graph;
}

// Find function with path compression
int find(struct Subset subsets[], int i)
{
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
```

```c
    return subsets[i].parent;
}

// Union by rank
void unionSets(struct Subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Comparison for qsort
int compareEdges(const void* a, const void* b)
{
    struct Edge* edgeA = (struct Edge*)a;
    struct Edge* edgeB = (struct Edge*)b;
    return edgeA->weight - edgeB->weight;
}

// Kruskal's algorithm
void kruskal(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V - 1];  // MST edges
    int edgeCount = 0, index = 0;

    // Sort edges by weight
    qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compareEdges);

    // Create subsets
    struct Subset* subsets = (struct Subset*)malloc(V * sizeof(struct Subset));
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Process edges
    while (edgeCount < V - 1 && index < graph->E)
    {
        struct Edge nextEdge = graph->edges[index++];

        int x = find(subsets, nextEdge.source);
        int y = find(subsets, nextEdge.destination);

        if (x != y)
        {
            result[edgeCount++] = nextEdge;
            unionSets(subsets, x, y);
        }
    }

    // Print MST
    printf("\nMinimum Spanning Tree:\n");
    for (int i = 0; i < edgeCount; ++i)
        printf("(%d, %d) Weight: %d\n", result[i].source, result[i].destination, result[i].weight);
```

```
    // Free allocated memory
    free(subsets);
}
```

```
Enter the number of vertices: 5
Enter the number of edges: 7
Enter edges (source destination weight):
0 1 4
0 2 3
1 2 2
1 3 1
2 3 5
2 4 4
3 4 6

Minimum Spanning Tree:
(1, 3) Weight: 1
(1, 2) Weight: 2
(0, 2) Weight: 3
(2, 4) Weight: 4

Process returned 0 (0x0)   execution time : 30.623 s
Press any key to continue.
```

```c
#include<stdio.h>
#define INFINITY 9999
#define MAX 10

void dijkstra(int G[MAX][MAX], int n, int startnode);

int main()
{
    int G[MAX][MAX], i, j, n, u;

    printf("Enter no. of vertices: ");
    scanf("%d", &n);

    printf("\nEnter the adjacency matrix:\n");
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n; j++)
        {
            scanf("%d", &G[i][j]);
        }
    }

    printf("\nEnter the starting node: ");
    scanf("%d", &u);

    dijkstra(G, n, u);

    return 0;
}

void dijkstra(int G[MAX][MAX], int n, int startnode)
{
    int cost[MAX][MAX], distance[MAX], predecessor[MAX];
    int visited[MAX], count, min_distance, nextnode, i, j;

    // Initialize cost matrix
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n; j++)
        {
            if(G[i][j] == 0)
                cost[i][j] = INFINITY;
            else
                cost[i][j] = G[i][j];
        }
    }

    // Initialize distances, predecessors, and visited nodes
    for(i = 0; i < n; i++)
    {
        distance[i] = cost[startnode][i];
        predecessor[i] = startnode;
        visited[i] = 0;
    }

    distance[startnode] = 0;
    visited[startnode] = 1;
    count = 1;

    // Find shortest path
    while(count < n - 1)
    {
        min_distance = INFINITY;

        // Find the nextnode with the smallest distance
        for(i = 0; i < n; i++)
```

```
        {
            if(distance[i] < min_distance && !visited[i])
            {
                min_distance = distance[i];
                nextnode = i;
            }
        }

        visited[nextnode] = 1;

        // Update distances
        for(i = 0; i < n; i++)
        {
            if(!visited[i] && (min_distance + cost[nextnode][i] < distance[i]))
            {
                distance[i] = min_distance + cost[nextnode][i];
                predecessor[i] = nextnode;
            }
        }

        count++;
    }

    // Print results
    for(i = 0; i < n; i++)
    {
        if(i != startnode)
        {
            if(distance[i] == INFINITY)
            {
                printf("\nNode %d is unreachable from node %d.", i, startnode);
            }
            else
            {
                printf("\nDistance of node %d = %d", i, distance[i]);
                printf("\nPath = %d", i);

                j = i;
                do
                {
                    j = predecessor[j];
                    printf(" <- %d", j);
                } while(j != startnode);
            }
        }
    }
}
```

C:\Users\hp\Downloads\Files1\prog10_dij.exe

```
Enter no. of vertices: 5

Enter the adjacency matrix:
0 2 0 1 0
2 0 3 2 0
0 3 0 0 1
1 2 0 0 4
0 0 1 4 0

Enter the starting node: 0

Distance of node 1 = 2
Path = 1 <- 0
Distance of node 2 = 5
Path = 2 <- 1 <- 0
Distance of node 3 = 1
Path = 3 <- 0
Distance of node 4 = 5
Path = 4 <- 3 <- 0
Process returned 0 (0x0)   execution time : 46.198 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <stdlib.h>
#define INFINITY 9999

struct Edge
{
    int source, destination, weight;
};

void bellmanFord(struct Edge edges[], int n, int e, int start);

int main()
{
    int n, e, start;

    printf("Enter the number of vertices: ");
    scanf("%d", &n);

    printf("Enter the number of edges: ");
    scanf("%d", &e);

    struct Edge *edges = (struct Edge *)malloc(e * sizeof(struct Edge));

    printf("Enter the edges (source destination weight):\n");
    for (int i = 0; i < e; ++i)
    {
        scanf("%d %d %d", &edges[i].source, &edges[i].destination, &edges[i].weight);
    }

    printf("Enter the starting vertex: ");
    scanf("%d", &start);

    bellmanFord(edges, n, e, start);

    free(edges);
    return 0;
}

void bellmanFord(struct Edge edges[], int n, int e, int start)
{
    int *distance = (int *)malloc(n * sizeof(int));

    // Initialize distances
    for (int i = 0; i < n; ++i)
    {
        distance[i] = INFINITY;
    }
    distance[start] = 0;

    // Relax edges repeatedly
    for (int i = 0; i < n - 1; ++i)
    {
        for (int j = 0; j < e; ++j)
        {
            int u = edges[j].source;
            int v = edges[j].destination;
            int weight = edges[j].weight;

            if (distance[u] != INFINITY && distance[u] + weight < distance[v])
            {
                distance[v] = distance[u] + weight;
            }
        }
    }

    // Check for negative weight cycles
```

```c
    for (int i = 0; i < e; ++i)
    {
        int u = edges[i].source;
        int v = edges[i].destination;
        int weight = edges[i].weight;

        if (distance[u] != INFINITY && distance[u] + weight < distance[v])
        {
            printf("Graph contains a negative weight cycle.\n");
            free(distance);
            return;
        }
    }

    // Print shortest distances
    printf("\nShortest distances from the starting vertex %d:\n", start);
    for (int i = 0; i < n; ++i)
    {
        if (distance[i] == INFINITY)
        {
            printf("Vertex %d: Unreachable\n", i);
        }
        else
        {
            printf("Vertex %d: %d\n", i, distance[i]);
        }
    }

    free(distance);
}
```

C:\Users\hp\Downloads\Files1\prog10_bell.exe

```
Enter the number of vertices: 5
Enter the number of edges: 8
Enter the edges (source destination weight):
0 1 4
0 2 5
1 2 -2
1 3 6
2 3 5
2 4 7
3 4 8
4 0 9
Enter the starting vertex: 0

Shortest distances from the starting vertex 0:
Vertex 0: 0
Vertex 1: 4
Vertex 2: 2
Vertex 3: 7
Vertex 4: 9

Process returned 0 (0x0)   execution time : 51.714 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <limits.h>


int matrixChainMultiplication(int dims[], int n)
{
    int dp[n][n];


    for (int i = 1; i < n; i++)
        dp[i][i] = 0;


    for (int len = 2; len <= n; len++)
    {
        for (int i = 1; i <= n - len; i++)
        {
            int j = i + len - 1;
            dp[i][j] = INT_MAX;
            for (int k = i; k < j; k++)
            {
                int cost = dp[i][k] + dp[k + 1][j] + dims[i - 1] * dims[k] * dims[j];
                if (cost < dp[i][j])
                    dp[i][j] = cost;
            }
        }
    }

    return dp[1][n - 1];
}

int main()
{
    int n;


    printf("Enter the number of matrices: ");
    scanf("%d", &n);

    int dims[n + 1];


    printf("Enter the dimensions of matrices:\n");
    for (int i = 0; i <= n; i++)
    {
        printf("Dimension %d: ", i);
        scanf("%d", &dims[i]);
    }


    int minMultiplications = matrixChainMultiplication(dims, n + 1);
    printf("Minimum number of scalar multiplications: %d\n", minMultiplications);

    return 0;
}
```

C:\Users\hp\Downloads\Files1\prog11.exe

```
Enter the number of matrices: 4
Enter the dimensions of matrices:
Dimension 0: 10
Dimension 1: 30
Dimension 2: 5
Dimension 3: 60
Dimension 4: 10
Minimum number of scalar multiplications: 5000

Process returned 0 (0x0)   execution time : 12.029 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <stdlib.h>

// Structure to represent an activity
struct Activity
{
    int start_time;
    int end_time;
};

// Function to sort activities by end time
int compareActivities(const void* a, const void* b)
{
    return ((struct Activity*)a)->end_time - ((struct Activity*)b)->end_time;
}

// Function to perform activity selection
void activitySelection(struct Activity activities[], int n)
{
    // Sort activities using qsort
    qsort(activities, n, sizeof(struct Activity), compareActivities);

    printf("Selected Activities:\n");

    // The first activity is always selected
    int i = 0;
    printf("(%d, %d) ", activities[i].start_time, activities[i].end_time);

    // Consider the rest of the activities
    for (int j = 1; j < n; j++)
    {
        // If start time >= end time of the last selected activity
        if (activities[j].start_time >= activities[i].end_time)
        {
            printf("(%d, %d) ", activities[j].start_time, activities[j].end_time);
            i = j;
        }
    }
    printf("\n");
}

int main()
{
    int n;

    printf("Enter the number of activities: ");
    scanf("%d", &n);

    // Handle edge case: No activities
    if (n <= 0)
    {
        printf("No activities to process.\n");
        return 0;
    }

    // Allocate memory for activities
    struct Activity* activities = (struct Activity*)malloc(n * sizeof(struct Activity));
    if (!activities)
    {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Input activities
    printf("Enter start and end times for each activity:\n");
    for (int i = 0; i < n; ++i)
```

```
    {
        printf("Activity %d: ", i + 1);
        scanf("%d %d", &activities[i].start_time, &activities[i].end_time);

        // Validate input: Ensure start time < end time
        if (activities[i].start_time >= activities[i].end_time)
        {
            printf("Invalid input: Start time must be less than end time.\n");
            free(activities);
            return 1;
        }
    }

    // Perform activity selection
    activitySelection(activities, n);

    // Free allocated memory
    free(activities);

    return 0;
}
```

C:\Users\hp\Downloads\Files1\prog12_activity.exe

```
Enter the number of activities: 5
Enter start and end times for each activity:
Activity 1: 1 4
Activity 2: 3 5
Activity 3: 0 6
Activity 4: 5 7
Activity 5: 8 9
Selected Activities:
(1, 4) (5, 7) (8, 9)

Process returned 0 (0x0)   execution time : 17.652 s
Press any key to continue.
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_TREE_HEIGHT 100
 // Define the structure for a Huffman tree node
 struct Node
 {
    char data;
    unsigned frequency;
    struct Node *left, *right;
 };
 // Function to create a new node
 struct Node* newNode(char data, unsigned frequency)
 {
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    temp->left = temp->right = NULL;
    temp->data = data;
    temp->frequency = frequency;
    return temp;
 }
 // Function to build the Huffman tree
 struct Node* buildHuffmanTree(char data[], int frequency[], int size)
 {
    struct Node *left, *right, *top;
    // Create an array to store nodes
    struct Node* nodes[MAX_TREE_HEIGHT];
    // Initialize the array with nodes for each character and its frequency
    for (int i = 0; i < size; ++i)
    {
        nodes[i] = newNode(data[i], frequency[i]);
    }
    int n = size;
    // Build the Huffman tree
    while (n > 1)
    {
        // Sort the array of nodes based on frequency
        for (int i = 0; i < n - 1; ++i)
        {
            for(int j = 0; j < n - i - 1; ++j)
            {
                if (nodes[j]->frequency > nodes[j + 1]->frequency)
                {
                    struct Node* temp = nodes[j];
                    nodes[j] = nodes[j + 1];
                    nodes[j + 1] = temp;
                }
            }
        }
        // Create a new internal node with the two smallest frequency nodes as children
        left = nodes[0];
        right = nodes[1];
        top = newNode('$', left->frequency + right->frequency);
        top->left = left;
        top->right = right;
        // Remove the two nodes with the smallest frequency
        nodes[0] = top;
        for (int i = 1; i < n - 1; ++i)
        {
            nodes[i] = nodes[i + 1];
        }
        n--;
    }
    return nodes[0];
 }
 // Function to print the Huffman codes
 void printCodes(struct Node* root, int arr[], int top)
```

```c
{
    if (root->left)
    {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }
    if (root->right)
    {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }
    if (!(root->left) && !(root->right))
    {
        printf("%c: ", root->data);
        for (int i = 0; i < top; ++i)
        {
            printf("%d", arr[i]);
        }
        printf("\n");
    }
}
int main()
{
    char data[MAX_TREE_HEIGHT];
    int frequency[MAX_TREE_HEIGHT];
    int n;
    // Get user input for the number of characters
    printf("Enter the number of characters: ");
    scanf("%d", &n);
    // Get user input for each character and its frequency
    for (int i = 0; i < n; ++i)
    {
        printf("Enter character %d: ", i + 1);
        scanf(" %c", &data[i]);
        printf("Enter frequency for character %c: ", data[i]);
        scanf("%d", &frequency[i]);
    }
    // Build the Huffman tree
    struct Node* root = buildHuffmanTree(data, frequency, n);
    // Print the Huffman codes
    int arr[MAX_TREE_HEIGHT], top = 0;
    printf("\nHuffman Codes:\n");
    printCodes(root, arr, top);
    return 0;
}
```

C:\Users\hp\Downloads\Files1\prog12_huff.exe

```
Enter the number of characters: 5
Enter character 1: A
Enter frequency for character A: 5
Enter character 2: B
Enter frequency for character B: 9
Enter character 3: C
Enter frequency for character C: 12
Enter character 4: D
Enter frequency for character D: 13
Enter character 5: E
Enter frequency for character E: 16

Huffman Codes:
C: 00
D: 01
A: 100
B: 101
E: 11

Process returned 0 (0x0)   execution time : 23.887 s
Press any key to continue.
```